Infrastructure Concepts

Jim Kowalkowski
Marc Paterno

December 9, 2004

Abstract

We explore some concepts on the design of software infrastructure for HEP physics event processing, based on our review of the CDF, DØ, and MiniBooNE experiments' software.

This is an excerpt from a larger document.

Contents

In this document (an excerpt from a larger infrastructure-related document to which we contributed) we present several tasks that a (physicist) user might want to do. The examples we have chosen are:

- analysis
- reconstruction
- trigger
- code behavior analysis

We do not intend to present formal *use cases*; we sketch only enough of a description of the task to make the necessary points clear. For each task, we describe the main features of the software of one or more experiments that we believe produced a successful solution. In some sections, we provide a short series of related tasks to illustrate different aspects of the experiments' solutions.

1 Analysis task

First consider a simple task of histogramming the transverse momentum of the leading p_T muon in each event. We start from a DST, and we do not want to perform new reconstruction. We want to find those muons, already reconstructed, that were identified by a specific version of the muon reconstruction algorithm and with a specific set of parameters used in the reconstruction and particle identification.

Each of the experiments we have worked with has the concept of a *framework module* whose purpose is analysis (as opposed to reconstruction, online filtering, or other tasks).

Framework module: A *framework module* a coherent body of code (an object) that operates on a physics event and responds to external stimuli (computer science "events") to perform various actions related to its single task. It performs its task without direct interaction with other framework modules. It can make use of *framework services*, defined later.

Framework module

A user writing such a module would be responsible for implementing the few necessary member functions for the module to perform its task.

One such member function is the module constructor. After construction the module should be in a functional state. It should have correct values for all its parameters and should not require later re-configuration before use. The parameters for the module should *not* be compiled into the code; it should be possible to inspect the parameter set *without* having an instance of the module—or the source code for the module—present. There should not be a two-phase (or multi-phase) configuration in which configuration is *begun* on construction but is *completed* only after another module has been constructed.

The CDF framework has the concept of a help system that tells the users the meaning and allowed values of various parameters used to configure a module. Users have found this very valuable. The CDF mechanism is more tightly bound to the module than we prefer; we prefer that the help system knows about *parameter sets*, but not about modules. We prefer that the code of the module is independent of the code of the help system, and *vice versa*. DØ has the concept of configuring a module with a single set of parameter values; a module can be re-configured by giving it a new set of parameters. This organization makes the reconfiguration logic simple to write and easy to understand. The framework is responsible for associating a specific parameter set with the appropriate module instance.

In our example, the creation of the histogram in which we will collect muon p_T values should be done in the constructor.[1] The parameters for the histogram (the number of bins, and the minimum and maximum values for the range) should not be specified in the code, but should be provided by the parameter set given to the constructor. Within the constructor, we query the parameter set for these values. The parameter set has a type-safe interface for retrieving the value associated with a given named parameter. The parameter set conveniently groups associated parameters at a level of granularity chosen by the developer of the module.

The second important member function of an analysis module is the *analyze event* function. The argument for this function is (a `const` reference or pointer to) an *event*.

Event: An event is an in-memory object database that can support insertions and queries, but that supports neither deletions nor modifications of objects already inserted. The "query language" is not general, but is instead tailored to the physicists' needs, and is expressed in the interface of the event class. Each event contains raw data and derived products (such as trigger output and reconstruction artifacts) related to a single beam crossing, or simulation thereof.

Event

In our example, we are considering the histogramming of muons. Let's consider several sub-examples.

In one case, we want to histogram only those muons from a specific algorithm, in a specific code version, configured with a specific set of parameters. We want to make sure our sample is not contaminated by the event data objects that are the product of any other "rogue algorithms."

[1] The business of how to interact with ROOT is complicated, and we will avoid the details here.

Event data object: An *event data object* is either a part of the raw data, of the simulation information, or of the output of a reconstruction algorithm. It has a unique identifier *within the event*.[2] Associated with each event datum (but not necessarily stored in the event) is an object that holds the provenance of that datum. Such provenance information includes the details of the configuration of the algorithm that made the event data object and the identifiers of other event data objects used as inputs for the reconstruction of the event data object. The provenance may include additional information. There is no "algorithm part" to an event data object.

Event data object

In another case, we want to histogram the output of several algorithms, each into its own histogram. In this case, we don't want to fix the number of histograms during construction of the module; we want to discover what algorithms have been run, and histogram their output, and record the configuration information for each algorithm from which we discover a reconstruction product.

In yet another case, we want to histogram the output of a specific algorithm, and want the newest approved-for-conference-use version of that algorithm found in the event, but reject versions that are "too old" or "too new."

To support all these uses, the event data must be associated with the full set of configuration information, and the event's query mechanism must be able to use all of, or any part of, that information as a constraint upon selection.

A third example member function is the *divulge statistics* function. This function serves as a signal to the module that it is time for the module to report its current state. Clearly a more detailed specification than this is necessary—we include it here primarily to illustrate that not all module member functions have to do with (physics) event processing.

Two additional module member functions are *end of run* and *end of job*. In our example we have no need of these functions, so we do not implement them.

2 Reconstruction task

There is a wide variety of types of reconstruction tasks. We will use as an example *missing E_T reconstruction*, because it seems to be among the simplest of reconstruction tasks.

We create a single module to perform this task. The constructor for this module is passed the same sort of parameter set object as was the analysis module from

[2]We see no need for this identifier to be *globally* unique and the cost of making it globally unique seems prohibitive.

section 1. For each event, the algorithm must use a particular set of event object instances as input. In this constructor we specify *not* which actual instance of event objects are to be used as input—since they do not yet exist, and change from event to event—but rather how the algorithm is to identify which calculation of calibrated calorimeter energies it will use and which vertex it will use. These specifications need to be sufficiently accurate to be unambiguously identify the event objects to be used as inputs for the missing E_T calculation algorithm. In each case we specify the type of the event data object and a description of the configuration of the module that created that object. During construction of the module, we also get a handle to any *framework service* we need—in this case the *calorimeter geometry service*, which we will make use of during reconstruction.

Framework service: To get access to a global resource, we use a framework service. Services manage initialization of, access to, and lifetimes of objects that provide non-event data—such as geometry information, and run conditions information. It may be that access to ROOT histograms, *etc.* should also be managed by a service. Our list is not exhaustive.

Framework service

For the analysis module of section 1 we discussed the *analyze event* function; for a reconstruction module we implement a *process event* function. The difference is that *process event* is passed a non-`const` reference or pointer to the event and is expected to modify the event by the insertion of a new event data object. In this function, we first get handles to the correct input objects: the container of calibrated calorimeter tower energies and the requested vertex. If either is missing, we create a missing E_T object that contains status information indicating that we have tried and failed to reconstruct the missing E_T, and also contains the reason for the failure.

If the required inputs are found, we then iterate through the collection of calorimeter tower energies (in the event data object we obtained above), and determine the directed energy vector from the vertex to the center of each hit tower, summing the x and y components. This requires use of the geometry service to determine the center of each tower in the collection of hit towers.

Geometry service: The *geometry service* is responsible for determining the identifier of the current run, for determining what survey information to use for the run, and for translating physical component identifiers to obtain geometry information about the identified detector component. It is independent of event data objects—it is possible to determine the number of ϕ segments in the hadronic calorimeter without having any event data present.

Geometry service

At the end of the iteration, we create a missing E_T event data object, with the appropriate data values, and insert it into the event. This object is labeled with several pieces of metadata:

- the identifiers of the calorimeter tower energy and vertex event objects;
- the identifiers of the parameter set used to configure this module;
- possibly other items.

The missing E_T object is issued its own unique (within the event) object identifier upon insertion into the event.

In the design of the MiniBooNE reconstruction model, we found that it was possible to automate nearly all of this identification and labeling process. Authors of reconstruction algorithms need to do almost nothing in order to have their reconstruction products fully identified.

3 Trigger task

The trigger system places what are probably the strongest constraints on the scheduling features of framework. CDF has the concept of a *trigger path*, which consists of a series of modules in a fixed order that act together to produce the portions of event reconstruction necessary for a specific trigger decision. However, at CDF testing a trigger path in simulator is not sufficient to understand its behavior, because previous flows affect the result—because inputs are not sufficiently specified. Modules in different paths appearing in different orders could give different results. In §6 we address the subject of scheduling in more detail.

Trigger path

4 Instrumenting the Framework

It has often been useful to measure the performance of reconstruction programs—for example, to determine the speed of each individual task, or to find the location of a memory leak. The module-based framework has made this easy to do, because such tasks are handled in a common place and require neither instrumenting of user code nor recompilation of the program.

5 Running the Program

Users at both CDF and DØ have complained about the difficulty of running their respective reconstruction programs. Few users have expert knowledge of the program, causing many to use ntuple-form output. Both experiments have a system that is very flexible and powerful. However, both systems would benefit from more attention to ease-of-use: presenting the configuration in a simpler fashion, and guiding the user through more limited choices to make *reasonable* configurations of the program. In both experiments the configuration is hierarchically defined, and each node in the hierarchy can be defined in its own file. While this is a good thing, it makes it difficult for the user to understand the result of the configuration. It is hard to know what combinations of modules are valid; the system does not help the user know what is valid. A browser of the hierarchy is a valuable tool; DØ has recently developed such a thing. But even this does not allow the user to understand what change in an "upstream" module will cause a change in the behavior of a "downstream" module. Both experiments record the configuration of the program at it was run, but both experiments lack a way to easily browse this configuration information, or to easily share it between users.

Both experiments suffer from a lack of distinction between the *build* environment and the *execution* environment. At DØ the execution environment was introduced as a late concept; tools (scripts) are provided to configure the environment and run the program. These tools are complex. At CDF there is no separation of the environments because a function (the constructor for the class `AppUser-Build`) is intended to be tailored by the user, in order to determine what modules are available for use, and what their instances' names should be.

6 Event Processing Schedule

One of the important functions of a framework is to build an event processing schedule. The schedule expresses what activities can be done in parallel and what must be done serially. The "milestones" are also identified (multiple activities must be completed before continuing). Many of the concepts listed below are necessary for building a good schedule, one that:

- uses resources efficiently,
- minimizes event processing time,
- eliminates redundant calculations,
- is easy to configure,
- gives consistent and proper results, and
- can alert the user if a configuration is invalid or has ambiguities.

The sequences, dataflows and decision points, multithreading, and input and output requirements can all be used in the creation of an event processing schedule. Removing one of these pieces of information from the problem list will likely mean that the schedule will be suboptimal. Are the four items listed above enough to create this schedule at run time? How are they expressed in the configuration of the program?

Inconsistent results is one problem that can occur if there is not enough information available during the schedule generation. A simple example illustrates a problem that can occur.

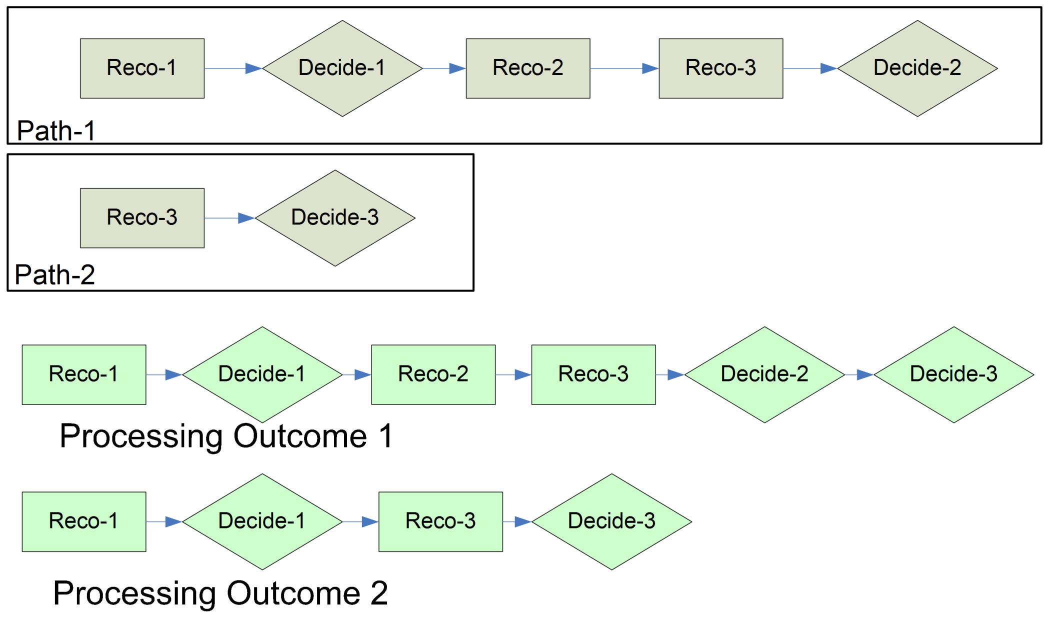

Figure 1: Example paths and possible execution sequences.

Figure 1 shows two paths that have the require the same instance of reconstruction to run in order to make a decision. The framework will ensure that this reconstruction only occurs once for each event. If the outcome of *Reco-3* depends on *Reco-2* results and this relationship is not enforced within the framework, then the outcome of *Decide-3* will have a hidden dependency on the outcome of *Decide-1*. To determine whether this set of paths produces a consistent result would require some kind of "coverage analysis," where data representing the entire range of inputs are fed through and the results are checked for consistency.

7 Dynamic Libraries

We want to discuss the use of dynamic (shared) libraries.

Does dead (unsed or test-related) code in a dynamically linked library cause performance problems because it is always present, unlike a static library where dead code costs little to nothing? It takes up virtual memory when the file is mapped. Is it true that a small amount of dead code is not really going to hurt much because of the demand paging mechanism in the operating system and the lazy symbol linking?

Should shared libraries used by an executable come from only one frozen release so they are always consistent? Is this a written procedure or a policy actually enforced in the program? How does this policy or procedure fit in with algorithm developers? Is it too restrictive? Are there different rules for developers versus production? Can the build produce a dependency database that is built into the programs and also punch versioning tags (library and release) into libraries to be used for validation? A scheme such as this could allow the dynamic library mix to be determined by program configuration at run time.

How does one insure that in critical applications the program will not fail after a period of running because a shared object library cannot be loaded after encountering an event that causes a new event processing path to be executed? One method is to only pull in libraries during program configuration.

Where is explicit loading and implicit loading used? Are only low-level libraries such as ROOT and persistency explicitly linked? Are all the reconstruction tasks implicitly linked?

8 Topics We Haven't Time to Write About

There are many topics we believe we need to discuss during our meeting, but about which we do not have time to write. We list them here with brief explanations, to help establish the scope of topics that we believe it is important to cover. We have ordered them in order of their importance, with the most important items first.

1. **INTER-OBJECT LINKS**: We believe it will be advantageous to make sure that the persistent form of inter-object pointers within an event is realized in terms of "dumb data" rather than any persistence mechanism's smart pointers. The CDF and D0 experiments placed restrictions on the use of pointers within persistent objects. Objects in an event could refer to other objects in the same event or to items heard within them. A reference to an item within an object

appears as a tuple of identifiers (EDOID,index), where EDOID is the event data object ID discussed earlier in this document and index is the item in this object in which we are interested. The event data object that supports indexing is required to supply a method that will produce the desired object given the index. Many persistent data objects fit this random access container pattern.

2. **SEQUENCES OF MODULES**: We believe it is useful for the framework to have a concept of *sequences* of modules that can be manipulated as a single unit. This organizational component helps reduce the complexities of understanding what a particular program configuration is doing.

3. **DATAFLOWS AND DECISION POINTS**: We believe that, in conjunction with sequences from item 2 above, that it is useful for the framework to have the concepts of *dataflows* and *decision points*, from which can be constructed trigger paths (among other purposes). A dataflow is a series of paths through which an event moves and a specification of the actions that occur within the paths. A path may have distinct stages such as merging several input streams, doing the reconstruction, tagging output results, or creating an analysis ntuple. A decision point is a place along a path where code must make some qualitative decision about the event. Whether or not the event will proceed down a path or be steered to other paths is determined by the configuration of a decision point.

   The information conveyed by these concepts can be used to build a static *schedule*. The information can be used to constrain an ensemble of module sequences and express how there will work together. Constraint examples can be "path one must come before path two" and "path three only happens if path two succeeds".

   We want to discuss whether it is sufficient to have a single "process" method that handles all types of actions performed on or with an event, or if we should have *several* different methods, each with a different purpose.

4. **CALIBRATION**: We want to make sure we discuss the requirements placed upon the framework and event model by the specialized task of generating and studying calibrations. There are also issues to be discussed in the subject of *using* calibrations.

5. **INTERACTIVE USE**: We would like to discuss the special demands placed upon the system by the need to support interactive use. This includes issues of a diverse natures, such as an interactive help system and interactive analysis.

6. **MULTITHREADING ISSUES**: We would like to discuss the possibility of taking greater advantage of the use of multithreading, for example in the parallel reconstruction of multiple events, and parallelism within the reconstruction of a *single* event. We believe this will become more important in the future, as multiple-processor machines and multiple-core processors become more common.

   Given the large amount of ancillary data necessary to process events and the large start-up time, there may be an advantage to starting up a single executable and create an event processing "pipeline." One image of geometry and calibration data can be established along with one instance of the data handling and persistency code. Independent paths (or subpaths) can be executed in parallel. We believe that such schemes may not require physicist-developed code to be multi-threaded. The EDM and framework will do the synchronization and resource scheduling.

7. **DATA INPUT AND OUTPUT**: We want to discuss the writing of multiple streams of output, merging multiple streams of input, and the usefulness of "tagging" events (and objects within events) in the creation and processing of multiple input and output streams.

8. **UNPACKING**: We believe it is useful for the core event model to provide utilities for the unpacking of "raw" data, in the interest of efficiency and ease-of-use.

9. **DATA VISUALIZATION**: Data visualization requires use of the interactive features of the framework. The tools used for this operation typically complicate the system because they have their own concept of an event loop. The data formats necessary to visualize the data can be quite different than those in an event. We would like to discuss how the interactive features of a framework allow for interfacing to visualization packages.

10. **HISTOGRAM AND NTUPLE SERVICE**: We recognize that ROOT will be used by many, if not all, CMS collaborators. We think it is likely that one of the common uses of the analysis framework will be to produce programs that make ROOT ntuples (or `TTree`s) specialized for a specific task. We also expect there to be a need for the use of histograms during reconstruction especially during the commissioning of the CMS detector. We believe that the mechanisms provided by ROOT for the management of histograms and ntuples are inadequate for serious use, and that a *histogram service* should be provided to handle the management.

11. **Specifying Inputs and Outputs**: Is it a good idea for modules to declare a list of necessary inputs and products? If so, how is this expressed? It is likely that object types is not good enough and that some of the EDM metadata is necessary (*e.g.*, algorithm name and version or configuration parameter values). How is the static declaration part expressed? If there is a dynamic component (one that uses metadata configuration), when does it need to be calculated? Is it before or after module construction time?

    Is this information and the dataflows and decision points information enough for the framework to produce a deterministic schedule? We desire a schedule that eliminates redundant work and minimizes the chances of inconsistent results due to order dependencies within a program configuration.

12. **Error handling**: We believe that the algorithms contributed by physicists and others that plug into the framework cannot determine directly know what actions are required as a result of an observed adverse condition. The context in which the program is running will dictate what the proper actions will be when these conditions occur. This implies that modules and algorithms report conditions and the framework determines the proper course of action[3] (*e.g.*, abort event processing, ignore, skip a portion of processing, abort a run, save the event for further testing, restart the program). We would like to discuss how these conditions are reported by user code and how the framework makes use of the information.

13. **Calibration and Alignment Data Management**: We believe it is important for it to be easy for users to obtain the appropriate calibration data and alignment data for a given data sample. We think it would be useful to discuss how the framework can automate this process.

[3]The framework should allow run-time configuration to determine the course of action to be taken when a module fails.